Algorithms for Network-on-Chip Design with Guaranteed QoS

Guy Even* Yaniv Fais

September 2, 2015

Abstract

We present algorithms that design NoCs with guaranteed quality of service. Given a topology, a mapping of tasks to processing elements, and traffic requirements between the tasks, the algorithm computes the interconnection widths, a detailed static routing, and a periodic scheduling. No headers, control messages, or acknowledgments are required.

The algorithm employs fractional Multi-Commodity Flow (MCF) that determines the widths of the interconnections as well as the routes of the flits. The MCF is rounded to a periodic TDM schedule which is translated to local periodic control of switches and network interfaces.

Our algorithm is applicable to large instances since every stage is efficient. The algorithm supports arbitrary topologies and traffic patterns. Routing along multiple paths is allowed in order to increase utilization and decrease latency.

We implemented the algorithm and tested it with the MCSL benchmark. Experiments demonstrate that our solution is stable and satisfies all the real-time constraints.

*School of Electrical Engineering, Tel-Aviv University, Email: guy@eng.tau.ac.il

1 Introduction

Networks-on-Chips (NoCs) have been proposed as a way to design Multi-Processor Systems-on-Chip (MPSoC) [6, 14]. NoCs constitute good engineering practice in the sense that they support modularity and, even more importantly, NoCs help decouple communication from computation and processing. NoCs are characterized by stringent real-time constraints and high reliability (packets can not be lost). Satisfying these requirements forces one to resort to short packets (called flits) and, even more importantly, use routing and scheduling without the layers of abstraction that characterize networks (such as TCP/IP).

A common paradigm in network design in general, and NoCs in particular, is the usage of virtual channels and dynamic routing [7]. Virtual channels provide an abstraction of end-to-end links which allow for time-division multiplexing (TDM) of the network interconnections. Dynamic routing offers flexibility in using the network resources. Successful online allocation of virtual channels cannot be guaranteed [2, 3]. Hence, online routing schemes cannot assure reliability let alone Quality-of-Service (QoS). As partial provision of bandwidth in real-time systems renders the system useless, the design of NoCs for a real-time applications requires prior knowledge of the traffic.

In this paper we focus on designing NoCs with respect to persistent traffic. Specifically, we address the following issues that are usually not addressed in theoretical papers: avoiding headers to reduce overhead, guaranteeing arrival of packets so that control messages (such as acknowledgments) are not needed, and guaranteeing real-time constraints.

1.1 Contribution

We present an efficient methodology for designing NoCs. The methodology can be applied to any NoC topology and deals in a unified fashion with the following algorithmic problems: assigning widths to interconnections in the NoC, computing routing, computing the scheduling, and computing the local controls of each network component. Our methodology can be used to automatically generate NoCs including detailed design of all the network components. Such a tool can help in implementing specific real-time applications over parallel platforms such as multi-processor systems on a chip.

The starting point is a traffic pattern that "dominates" the actual traffic to be supported. The first stage of our algorithm solves a fractional multicommodity flow (MCF) problem. The MCF formulation is used to compute the required edge capacities and is easily adapted to different objective functions such as: reducing total wire cost, bounding the capacity of links, and limiting lengths of flow paths. The computed MCF determines edges capacities as well as the routing of the requests. A periodic schedule that supports

the MCF is computed in the next stage. This periodic schedule is then implemented by programming the control of the nodes in the network.

In NoCs, the common practice is to send (long) messages by long trains of contiguous flits [7]. We deviate from this practice because such long trains block the network, incur large delays, and reduce utilization. Instead, we employ schedules in which flits belonging to the same message can be sent in non-contiguous sets of time slots and even along different paths.

We show that periodic schedules can be implemented without any headers, control messages, nor acknowledges. Our synchronous implementation delivers all the packets (no packet is dropped) and does not require "warming up" of the network. We guarantees safe delivery of every packet. Our implementation suggests a trade-off in which local control (an abundant resource in modern VLSI chips) can help save in wires (a nonscalable resource).

To increase network utilization we remove the restriction that all packets of the same session must traverse the same path. Splitting flow into multiple paths helps balance the load, but is considered unfavorable due to packets arriving out-of-order. Our implementation of periodic schedules reorders the packets without requiring the attachment of serial numbers to packets.

We also suggest a method for designing NoCs for real time systems. The real-time system is modeled by a constraints task communication graph (TCG) that specifies dependencies between tasks. In addition, the time required to complete each task (once its inputs are available) and the length of the messages between tasks is also part of the TCG. We suggest to bound the delay of each message by a linear function of the message length. Thus, the choice of the linear function of the delay bound implies deadlines for all tasks, and we obtain a real-time benchmark that can be easily checked.

We show how to reduce the delay bounds of the messages in the TCG to a multi-commodity flow problem. This reduction enables us to employ our NoC design methodology to support real-time systems. Our simulations suggest that the methodology yields NoC designs that meet the deadlines, and thus real-time performance is achieved.

1.2 Previous Work

Multi-commodity flows were previously used to design NoCs. In [5,12] multi-commodity flow was used for computing mappings of cores in NoC architectures. Schoeberl *et al.* [13] employed an integer multi-commodity flow for designing NoCs with uniform traffic. Murali *et al.* [11] solve a fractional MCF and randomly round the fractional flow to obtain an unsplittable flow solution. Hu *et al.* [8] employ MCF to reduce the power consumption of NoCs. Hu *et al.* [9] employ MCF to find the best topology from a given

set for a latency-power product objective.

Sørensen *et al.* [15] present a meta-heuristic scheduler for inter-processor communication in multi-core platforms using time division multiplexed (TDM) NoCs. Given a specification of the target platform and of the application, the algorithm generates a periodic TDM schedule. The algorithm is based on solving an integer multicommodity flow problem. Since this problem is NP-Complete, the running time of the algorithm does not scale well. In addition to designing a more general scheduler, the goal of minimizing the period of the schedule is obtained by increasing the frequency of the TDM clock. The results obtained in [15] for the MCSL benchmark on the 8×8 torus indicate[1] that limiting the TDM period to 100 time slots requires increasing the TDM clock frequency by a factor of 10 to 100.

Andrews et. al [1] prove bounds on packet delays for periodic scheduling in store-and-forward packet routing networks. (We elaborate on their setting in Sec. 3.) They proved the existence of a periodic schedule with an asymptotic optimal delay bound. They also present a simple randomized algorithm in which the asymptotic delay bound is bigger by a logarithmic factor. Their setting assumes that paths and rates for all sessions are given as well as a positive constant slack in the congestion of the edges. The result of Andrews et. [1] justifies focusing on periodic schedules in this setting.

1.3 Organization

In Sec. 2, we briefly describe what NoCs are. In Sec. 3, we discuss the token based periodic schedules of Andrews et. al [1]. In Sec. 4, we propose a parametrized timing specification of real-time systems. In Sec. 5, we show how to implement periodic schedules without requiring any headers. In Sec. 6, we present our NoC design methodology. In Sec. 7, we present a reduction of real-time task communication graphs to multi-commodity demands. Experimental results are described in Sec. 8. Further directions of research are discussed in Sec. 9.

2 What is a Network-on-Chip?

Networks-on-Chips (NoCs) are store-and-forward packet networks that support communication between modules on a chip (e.g., multiple cores). The idea is to replace dedicated point-to-point interconnections between the modules as well as buses by a packet forwarding network. As in regular data networks, the NoC hides the details of how data is sent between modules.

[1] Our guess is that the results were reported for a mesh with links 32-bits wide.

A NoC can be modeled by a graph $G_N = (P \cup V, E)$, where P denotes the set of processing elements (PEs) and V denotes the set of NoC switches. Each PE is connected to the NoC via a dedicated network interface (NI), hence we identify each PE with its network interface.

The design of a NoC involves classical issues in parallel computation and networking, including: choosing the network topology, mapping tasks to PEs, computing the routes of packets, scheduling packets, managing buffers, etc.

3 Periodic Schedules Based on Token Sequences

In this section we overview the periodic schedules that are computed for store-and-forward packet routing networks in Andrews et. al [1]. We are given a synchronous network N with arbitrary topology. The goal is to route sessions over N. Each session i consists of a source node a_i, a destination node b_i, a simple path p_i from a_i to b_i, and an injection rate r_i. The injection rate limits the number of packets that may be injected by session i. Formally, for every t, at most $r_i \cdot t + 1$ packets may be injected by session i during every interval of length t. The sessions must satisfy the following congestion property. There exists an $\varepsilon > 0$ such that for every edge e, $\sum_{p_i \ni e} r_i \leq 1 - \varepsilon$.

In the language of multicommodity flows, each session is a commodity with demand r_i. All edges have unit capacity, flow is not splittable, and the routing is given. The algorithm requires that the load of every edge may not exceed $1 - \varepsilon$.

A template-based periodic schedule is suggested as a basis for the scheduling algorithm in [1]. The terminology of this schedule consists of a *period*, *tokens*, *token sequences*, and *templates*. Let Φ denote the *period* of the schedule. Each directed edge in the network is attached a *template* that is a table with Φ entries. Each entry in a template is called a *token* and its value is a serial number of a token sequence. A *token sequence* for session i is an allocation of one token in each template of the edges along the path p_i. The token sequence is used for sending packets at a rate of one packet per period from the source a_i to the destination b_i. To support the demand of session i, session i should have $r_i \cdot \Phi$ token sequences assigned to it.

The rule for scheduling packets according to a template is as follows. Consider the packets of session i. The packets wait in a FIFO input queue of session i in the source node a_i. Let e_1 denote the first edge of p_i. Whenever a time slot occurs in which the token in the template of e_1 is a token sequence of session i, the first packet in the input queue of session i is dequeued, and sent along the edge e_1. From this point on, each further hop of the packet may only use the same token sequence used for the first hop. This means that the incoming packet is stored in the interior node together with its token sequence.

This packet is sent along the next edge of p_i as soon as a time slot arrives whose token equals the token sequence of the packet.

A naive implementation of the scheduling algorithm requires that each packet carry the token sequence that was used for its first hop (in Sec. 5 we show that this is not required). Alternatively, [1] present a distributed randomized scheduling algorithm. The distributed algorithm delays each token independently and uniformly. Each packet is preassigned a deadline for each hop. Contention is resolved by employing an earliest-deadline-first policy (EDF). The distributed algorithm does not require templates. However, the header of each packet must contain an encoding of its path (such as its session number so that each intermediate node can determine the next hop) as well as its random delay (so that EDF can applied).

4 Timing Specification of Real-Time Systems

In this section we propose a parameterized timing specification of a real-time system. The specification is based on a precedence graph over the tasks and bounds the delay of messages by a linear function. This specification is used as a benchmark in the evaluation of real-time applications implemented over NoCs.

Task Communication Graphs (TCGs). Real-time systems can be described by a precedence graph over tasks. Dependencies between tasks are derived from the communication between the tasks. The precedence graph is a directed acyclic graph, called a *task communication graph*. Each vertex in the TCG corresponds to a task and is labeled by the time it takes the task to complete once all the inputs are ready. An arc from task u_i to task u_j means that task u_i sends a message to task u_j. A task u_j cannot begin before all the messages from incoming edges arrive. Each arc in the TCG is labeled by the length of the message. External inputs are modeled by sources in the TCG and external outputs are modeled by sinks. Each source is labeled by an *arrival time* that specifies when this input is injected into the system. Each sink is labeled by a *deadline* that specifies the time by which it should receive its incoming message (which is the output of the system). Real-time compression of video from a camera is a canonical example of a real-time system. Arrival times are determined by the rate in which raw data from the camera is fed, and deadlines of compressed frames are determined by the maximum allowed delay.

Communication delay bounds: from edges to tasks. We propose to model the delay of a message by a linear function. Let $|m|$ denote the length of a message, L

the latency parameter, and α denote the width of the edge e. The delay bound of a message m sent along e is at most $L + |m|/\alpha$ time slots. Note that higher values of L or lower values of α allow more time for communication, and thus postpone deadline. The completion times of each task are computed by dynamic programming based on the task communication graph, message lengths, computation duration of each task, input arrival times, and the communication delay parameters L and α.

Satisfying real-time specification. A pair L and α is *feasible* if the completion time of the sinks does not exceed their deadlines. In the absence of deadlines in sinks (as well as intermediate nodes), the completion times of such a hypothetical network serve as our benchmark. Namely, we define the deadlines to be the completion times of the tasks if each message m is delivered after $L + |m|/\alpha$ time slots. The role of the parameters L and α is to obtain deadlines if we have none, and to help in the reduction of the TCG specification to a multi-commodity flow problem (see Sec. 7). Our goal is to design a NoC that delivers the same performance as such a hypothetical network. As α increases and L decreases, the challenge in designing a NoC is harder because the "competing" hypothetical network has a smaller communication delay.

A system meets the real-time specification if the outputs satisfy the deadline constraints (that depend on L and α). The *lag* of a task is the difference between the time in which the task ends and the time in which it is supposed to end. Positive lags correspond to violations of deadlines. In fact, bounded lags (that do not grow as a function of time) are usually tolerable. Unbounded lags that increase linearly over time, often called *drifts*, and mean that the system is running too slowly. Elimination of drifts (using the same network) requires accelerating the clock rate of the network.

5 Implementation of Periodic Schedules

In this section we show how to implement the switches and network interfaces in a NoC so that they execute a given periodic schedule. This implementation does not require any headers. In particular, a packet does not need to carry its path, session number, serial number, etc.

5.1 Representation

We first describe how a periodic schedule is represented. One option is to have a schedule based on token sequences. Another option is that the periodic schedule is represented by a set of periodic schedules, one per edge. Let S denote the set of sessions and let σ_e denote

the schedule for edge e. The schedule of an edge e is a function $\sigma_e : \{0, \ldots, \Phi - 1\} \to S$ that specifies which session is scheduled in each time slot.

Reduction to token sequences. We reduce a periodic schedule that is represented by edge schedules to token sequences by matching incoming slots and outgoing slots of each session. By following time slots that are matched along the flow paths, we can "decompose" the flow of packets of each session into token sequences. We note that such matchings are possible because flow must be conserved in each intermediate node. Namely, the number of time slots allocated to s along incoming edges equals (or is at most) the number of time slots allocated to s along outgoing edges. Hence such a matching exists. In this matching it is desirable to minimize the delay from the source to destination of session s. If session s uses a single path, then the matching is based on a simple first-in-first-out policy. We point out that computing matchings that minimize end-to-end delay for splittable flows is an interesting problem.

5.2 Implementation

The proposed NoC architecture (which is similar to [12, 15]) consists of processing elements (PEs), network interfaces, switches (with small buffers and local control) and interconnections (whose widths are determined by the algorithm). Each PE is connected to a switch via a Network Interface (NI) (for simplicity, we allow each NI to be connected only to a single switch).

Once we have a representation by token sequences, each switch node v is implemented by a memory and a control as follows. The memory can store $\Phi \times d_{\mathsf{in}}$ flits, where d_{in} is the number of incoming edges.[2] In time slot i, the memory stores all the incoming flits in the ith row of the memory. Each outgoing edge e is controlled by a function $f_e : \{0, \ldots, \Phi - 1\} \to \{0, \ldots, \Phi - 1\} \times \{1 \ldots, d_{\mathsf{in}}\}$. The value of $f_e(t)$ specifies the memory address that contains the flit to be sent along edge e in time slot t. Note that a concurrent read and write to the same location mean that the read returns the value before the write (because a flit cannot arrive and be forwarded in the same time slot).

Network interfaces (NIs) are sources and sinks of sessions (but not intermediate nodes). An NI contains a queue for each session. If the NI is the source of the session, then it dequeues a packets whenever the time slot of the outgoing edge is allocated to the session, and sends the packet along this edge. If the NI is the destination of the session, then it adds the arriving packet to the queue.

[2] In our implementation, we reused memory if a flit stays in the memory for only one time slot.

5.3 Reordering of Packets

If a session is routed along multiple paths, then the arrival order of the packets may not equal the order in which they have been sent. We describe now how the network interface of the destination can reorder the packets without having to attach serial numbers to the packets.

As described above, the NoC implements the policy dictated by a periodic schedule that is represented by token sequences. Consider session s and assume that the schedule has k token sequences for session s. Suppose we number the token sequences from 1 to k, and let t_i denote the time slot in which token sequence i leaves the source. We assume that $t_1 \leq t_2 \leq \cdots \leq t_k$. (If multiple token sequences leave the source in the same time slot, then we order them arbitrarily, but the source must send the packets according to this order.) The destination network interface of session i reorders the incoming packets as follows. Note that packets that are delivered using the same token sequence (but in different periods) arrive in order. Hence, the sorting needs only to deal with merging k sorted sets. Let D_i denote the accumulated delay of token sequence i. This means that a packet starts moving in time slot $\Phi \cdot \ell + t_i$ must arrive at time slot $\Phi \cdot \ell + t_i + D_i$. Hence the difference in time slots between the arrival of consecutive packets from token sequences i and $i+1$ equals $t_{i+1} + D_{i+1} - (t_i + D_i)$. As this difference is fixed, the network interface can easily merge the packets from different token sorted in the correct order.

6 NoC Design Methodology

In this section we present a methodology for designing NoCs. The starting point is the NoC topology and multi-commodity demands over the processing elements of the NoC. The algorithm computes the widths of the interconnections in the NoC and the periodic schedules of NoC component (switches and network interfaces). Based on these widths and schedules, the network can be automatically generated. The algorithm proceeds in the following phases.

Fractional multicommodity flow (MCF) formulation. The MCF instance is over the NoC graph $G_N = (P \cup V, E)$ with demands $\mathrm{bw}(PE_i, PE_i)$ between pairs of PEs. The demands describe the traffic rates between the PEs. The algorithm computes a flow $f_{(PE_i,PE_j)}$ over the NoC graph for each ordered pair (PE_i, PE_j). The flow amount of $f_{(PE_i,PE_j)}$ equals the demand $\mathrm{bw}(PE_i, PE_j)$. Let $f(e) \triangleq \sum_{i,j} f_{(PE_i,PE_j)}(e)$ denote the total flow along e. The width of the interconnection e is proportional to $f(e)$. Linear programming formulations of the MCF are quite flexible, so we describe three possible objectives.

1. Minimize total interconnection cost. In this formulation each edge e has a cost $c(e)$ that corresponds to its length. The goal is to minimize $\sum_e c(e) \cdot f(e)$. In fact, this instance uses shortest paths routing and can be solved without linear programming. The solution is simply an overlay of shortest paths of all the demands.

2. Minimize maximum congestion along edges. In this formulation each edge e has a width $u(e)$. We add constraints of the form $f(e) \leq \lambda \cdot u(e)$ for each edge. The objective is to minimize λ.

3. Limit the lengths of flow paths. The length of a flow path is a heuristic indication of the cost and delay associated with the path.

We remark that linear programming solvers can easily solve instances with thousands of commodities.

Rounding of flows. Let Φ denote the requested period of schedule.[3] If flow is measured in flits per time slots, then the smallest unit of flow (i.e., granularity) is $1/\Phi$. This means that we need to round the flows of each pair to multiples of $1/\Phi$ as follows. For each pair (PE_i, PE_j), decompose the flow $f_{(PE_i,PE_j)}$ into flow paths $p_1, p_2, \ldots$. Assume that the paths are sorted in descending flow amount, namely, $f_{(PE_i,PE_j)}(p_k) \geq f_{(PE_i,PE_j)}(p_{k+1})$. Initialize $d(i,j)$ to equal the demand $\mathrm{bw}(PE_i, PE_j)$. Scan the paths $\{p_k\}_k$ starting with p_1. If $d(i,j) > 0$, round up $f_{(PE_i,PE_j)}(p_k)$ to a multiple of $1/\Phi$ and update $d(i,j) \leftarrow d(i,j) - f_{(PE_i,PE_j)}(p_k)$. If $d(i,j) = 0$, then the remaining flow paths are erased.

Determine interconnection widths. If flow is measured in flits per time slots, then the width $w(e)$ (in bits) of each interconnection e in the NoC graph is simply $\lceil f(e) \rceil \cdot k$, where k denotes the flit size. Thus the width of each interconnection is big enough to accommodate the planned flow along it and there is no need to split flits.

Periodic Scheduling. A greedy allocation of time slots to demand along the edges is guaranteed to work. One can apply the algorithm of Andrews et. al [1] to compute a periodic schedule with proven bounds on the delay of each packet.

7 Reduction From TCG to Multi-Commodity Flow

In this section we present a reduction of a specification of a TCG of a real-time system to a multi-commodity flow problem. This reduction enables us to employ our NoC design

[3] There is a trade-off in the choice of Φ. The larger Φ is, the finer granularity we obtain, and thus rounding incurs a smaller overhead. On the other hand, the cost of the NoC switches increases as Φ grows.

methodology to support real-time systems.

Consider a TCG $G_T = (T, E_T)$ over a set of tasks T. The edges are labeled by the length of the messages. The delay bound of a message m_e that corresponds to e is $L + |m_e|/\alpha$, where L denotes the latency parameter and α denotes the width of the virtual channel.

Our goal is to define a multi-commodity flow (MCF) problem for a NoC that is supposed to support the execution of the real-time system modeled by the TCG G_T. The NoC is a graph $G_N = (P \cup V, E)$, where P denotes the set of processing elements (PEs) and V denotes the set of NoC switches (we identify each PE with its network interface, so we do not need to model network interfaces here). We are given a mapping $\pi : T \to P$ that assigns each task to a PE.

The MCF demands are set of demands between pairs of PEs. Namely, $\mathrm{bw} : P \times P \to \mathbb{R}$, where $\mathrm{bw}(PE_i, PE_j)$ denotes the maximum rate of traffic from the processing element PE_i to PE_j. Fix PE_i and PE_j. We suggest to define the value of $\mathrm{bw}(PE_i, PE_j)$ as follows. Consider all the edges $e = (\tau_a, \tau_b)$ in the TCG such that $\pi(\tau_a) = PE_i$ and $\pi(\tau_b) = PE_j$. Each such edge e induces a flow demand f_e from PE_i to PE_j during an interval I_e. The interval I_e starts when τ_a is completed, and its length is $L + |m_e|/\alpha$, where $|m_e|$ is the length of the message sent along e. The flow amount f_e equals $|m_e|/\alpha$. For every time t, let $f(t)$ equal the sum of the induced flow demands f_e for which $t \in I_e$. We define $\mathrm{bw}(PE_i, PE_j)$ to be $\max_t f(t)$.

After defining the MCF demands, we may define the objective function of the MCF according to Sec. 7, and apply the NoC design methodology.

8 Experimental Results

8.1 Benchmarks

Random steady traffic. In the random traffic model we randomly choose pairs of demands. The flit size is 4 bits and the demand $\mathrm{bw}(P_i, P_j)$ (in flits) is chosen uniformly from $\{0, \ldots, 5\}$. Hence, on average, a virtual channel delivers 10 bits per period.

The main purpose of the random steady benchmark is to test whether: (1) the algorithm is efficient for large instances, (2) the algorithm computes routings with a reasonable number of wires, and (3) the algorithm computes schedules with high utilizations, low latency, and small switch buffers.

The MCSL Benchmark [10]. The recorded traffic in the MCSL benchmark is based on mapping a multi-threaded program to multiple cores (PEs). The benchmark contains

TCGs of applications such as decoding and encoding of Reed-Solomon codes, FFT, etc. We used TCGs that were obtained from embeddings in 8×8 meshes. We emphasize that the TCGs of the MCSL benchmark do not include deadlines of external outputs or arrival times of external inputs. This benchmark enables one to perform detailed comparison between the real-time specification (deadlines of tasks) and the completion times of tasks in the NoC-based system.

8.2 Algorithm Implementation

The algorithm uses the CLP COIN-OR LP solver `https://projects.coin-or.org/Clp`. Even though the MCF objective in our experiment is the sum of edge costs, we used the LP solver to verify scalability. The algorithm is implemented in C++.

8.3 Simulator

Our simulator is based on the HNoCS [4] which uses the OMNET++ network simulator. This simulation is cycle accurate. We implemented generators for several parameterized topologies (3-level Clos, Beneš, Fat Tree - Butterfly with duplication of nodes, K-ary N-fly [5, 16], K-ary-N-Fly-Tree, and a random graph). We implemented modules for the network interfaces and the switches. We also implemented modules that feed the traffic from the benchmarks to the network interfaces. Simulation results are logged by monitoring modules.

8.4 Results

Random steady traffic. We executed a random traffic model with steady traffic for 16 to 256 PEs over various topologies[4]. The schedule period is 8 time slots. The results of the simulation are listed in Table 1. Basic floor-planning (for the topologies that are not the mesh) was employed to estimate the length of each interconnection as follows. The PEs are organized in a square matrix, and each switch is located in the "center of mass" induced by its distance to the PEs it communicates with. The Euclidean distance between the locations of the endpoints of an interconnection is used as an estimate of the interconnection length. This estimate of the length of interconnections is used as their cost for the MCF. The computed width of the interconnection is measured in bits (interconnections must be a multiple of the flit size). The cost of an interconnection is the product of its length and width. The buffers in the switches are referred to as the

[4]The random NoC graph was obtained by a union of two perfect random matchings altered to obtain connectivity.

Topology	Cores	Switches	Inter-connects	#Virtual Chan-nels	Avg Wire Length	Avg Wire Width	Total Wire Width	Total Wire Cost	Avg Wire Cost	Total Mem	Avg Mem	Avg Latency	Max Latency	Total Util %
Mesh	16	16	24	30	1.00	13.83	664	664.00	13.83	183	11.44	3.11	6	79.62
Clos	16	25	24	30	2.47	18.67	896	2116.72	44.10	360	14.40	4.06	6	92.83
Beneš	16	36	48	30	1.24	13.41	1287	2507.72	26.12	462	12.83	5.83	7	80.45
K-ary-N-fly	16	28	32	30	1.85	15.41	986	2202.40	34.48	438	15.64	4.42	10	85
K-ary-N-fly-Tree	16	24	32	30	1.85	13.38	856	2076.72	32.45	351	14.63	3.88	6	90.69
Random	16	16	29	30	2.24	8.41	488	1001.86	17.27	108	6.75	2.27	6	81
Fat-Tree	27	54	81	30	1.53	6.92	1121	2366.19	14.61	364	6.74	5.22	7	83.43
Mesh	64	64	112	188	1.00	22.39	5016	5016.00	22.39	1425	22.27	5.19	16	78.24
Clos	64	81	80	188	5.42	24.69	3950	15395.50	96.22	1906	23.53	4.17	11	69.73
K-ary-N-fly-Tree	64	80	128	188	3.38	15.12	3871	15316.50	59.83	1963	24.54	4.00	8	75.44
Mesh	144	144	264	1920	1.00	73.07	38583	38583.00	73.07	11038	76.65	8.60	31	79.39
Clos	144	169	168	1920	8.43	56.82	19093	104965.00	312.40	9935	58.79	4.24	11	75.43
K-ary-N-fly-Tree	144	168	288	1920	4.92	32.44	18686	104552.00	181.51	10570	62.92	4.16	10	81.93
Fat-Tree	256	512	1024	3155	3.22	2.63	5381	20969.60	10.24	2064	4.03	5.60	12	77.37

Table 1: Results with steady random traffic

memory and their size is counted in bits. The latency of a flit between NIs is given in time slots (i.e., clock cycles of the NoC). The average utilization is the fraction of time slots in which a message is delivered during the simulation (in the steady state).

These results demonstrate the flexibility of the algorithm with respect to various topologies and many PEs. The running time of the algorithm for 64 cores is about two minutes. The average latency increases moderately as the number of cores increases. For example, the average latency in the 4×4 mesh is 3.11 time slots compared to a latency of 8.6 in the 12×12 mesh. Namely, the average latency grows at a lower rate than the diameter. In small networks, the mesh wins in all parameters, however, in larger networks, the mesh has a higher latency (although it remains cheaper as there are fewer switches).

MCSL Recorded Traffic. We executed the algorithm for the 8 MCSL recorded traffic patterns over an 8×8 mesh. The period of the TDM schedule is 8 time slots. The real-time specification was derived from the TCG combined with an end-to-end delay bound of $L = 0$ (except for the FFT benchmark for which $L = 8$ was used) and a range of widths $\alpha \in \{4, 8, 16, \infty\}$ bits per second ($\alpha = \infty$ means that the real-time deadlines assume that any number of flits can be sent simultaneously over every interconnection). The results of the simulations are listed in Table 2. The results demonstrate that, except for the FFT benchmark, the NoC supports the traffic patterns with reasonable wire widths, buffer sizes, and latencies. We elaborate below on the very low utilization of time slots which is caused by the bursty nature of the traffic patterns.

It is interesting to note the following remarks on the simulation results: (1) The rounding of the fractional MCF to time slots and flits incurs an overhead of 5%-10%. This means that the rounding employed by the algorithm is rather efficient, and that the cost of the solution with respect to the maximum bandwidth requirements is at most 10% higher than the optimal cost (for the benchmark patterns). (2) Increasing the end-to-end

PROGRAM	α	Average Wire Width	Total Wire Width	Total Schedule Size	Total Memory	Avg Memory	Average Latency	Max Latency	Total Utilization
RS-32_28_8_enc	∞	0.87	194	1292	97	1.52	3.92	10	10.65
RS-32_28_8_dec	16	12.94	2898	22198	1495	23.36	3.67	9	12.24
Sparse	16	4.06	910	7008	373	5.83	3.96	16	7.79
Robot	8	2.54	570	3975	294	4.59	3.40	8	2.27
Fpppp	4	21.26	4762	32542	2404	37.56	5.45	26	0.95
H264-1080p_dec	4	20.44	4579	34562	2802	43.78	4.02	12	1.44
H264-720p_dec	4	14.86	3329	25768	1724	26.94	4.34	13	3.24
FFT-1024_complex[a]	4	76.55	17148	129544	10553	164.89	6.23	21	3.23

[a]Due to the complexity of the FFT benchmark, the value of the end-to-end delay bound L is set to 8. In the other programs $L = 0$.

Table 2: Results with MCSL recorded traffic over an 8×8 mesh with schedules of 8 time slots per period

From	To	High usage slots	Low usage slots	$Ratio$
0	1	1476952	0	infty
3	28	20226	22780547	0.0008

Table 3: Results with MCSL recorded traffic over an 8×8 mesh with schedules of 8 time slots per period.

delay L of virtual links used to derive the real-time specification has a small effect on the results of the algorithm. Only in the FFT benchmark application did we use an end-to-end delay of $L = 8$; in all the rest of the benchmark applications we used $L = 0$. (3) The real-time specification (based on the task communication graph, the running time of each task, the lengths of the messages, and the parameters L and α) defines the required end times of every task. The simulation (based on the execution of the tasks, the NoCs, and the periodic schedule) provides the end time of every task. Thus, we can compute the lag of every task. The sum of the lags over all the tasks (as well as the lag of the last task) is negative for all the simulations listed in Table 2. In addition, the maximum lag is small. Hence, we conclude that the simulated performance satisfies the real-time specification throughout the execution of the applications in the benchmarks.

Low Utilization with the MCSL Recorded Traffic. The ratio between the utilized time slots and the total time slots is very small in the simulations listed in Table 2. The algorithm computes a static routing, and therefore, to satisfy the real-time constraints must allocate bandwidth according to worst case traffic. On the other hand, the benchmark is very bursty. Indeed, in this setting, the utilization is upper bounded as follows:

$$\text{utilization} \leq \frac{\sum_{i,j} \text{average}(\text{bw}(PE_i, PE_j))}{\sum_{i,j} \max(\text{bw}(PE_i, PE_j))}$$

In Table 3, we compare the utilization obtained during the simulations of the NoC generated by the algorithm and the upper bounds on the utilization. This comparison shows that the low utilization can not be avoided by static schedulers that satisfy the real-time constraints of the MCSL recorded benchmarks.

9 Further Work

Static routing does not adapt to changes in traffic. Three possible directions for resolving this problem are: (1) Introduce a network manager entity that gathers information about the over- and under-utilizations of allocated bandwidth. New routings and schedules can be computed from time to time by the network manager based on the gathered information. The new schedules can be used to reconfigure the switches and the NIs. Note that in such a setting, NoC link widths are fixed, although one can re-orient wires of interconnections. (2) Overlay dynamic routing over the static routing. The idea is to mimic the way people use static schedules of trains for their dynamic needs. (3) Increase interconnection widths so that there are plenty of time slots that are not allocated by the algorithm. These vacant time slots can be used for a second dynamic routing protocol over the same network. Such a hybrid solution can encapsulate the static and dynamic networks so that they do not mix although they use the same infrastructure.

References

[1] Matthew Andrews, Antonio Fernández, Mor Harchol-Balter, Tom Leighton, and Lisa Zhang. General dynamic routing with per-packet delay guarantees of O(distance+ 1/session rate). *SIAM Journal on Computing*, 30(5):1594–1623, 2000.

[2] James Aspnes, Yossi Azar, Amos Fiat, Serge Plotkin, and Orli Waarts. On-line routing of virtual circuits with applications to load balancing and machine scheduling. *Journal of the ACM (JACM)*, 44(3):486–504, 1997.

[3] Baruch Awerbuch, Yossi Azar, and Serge Plotkin. Throughput-competitive on-line routing. In *34th FOCS*, pages 32–40. IEEE, 1993.

[4] Yaniv Ben-Itzhak, Eitan Zahavi, Israel Cidon, et al. HNoCS: Modular open-source simulator for heterogeneous NoCs. In *International Conference on Embedded Computer Systems (SAMOS)*, pages 51–57. IEEE, 2012.

[5] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *Parallel and Distributed Systems, IEEE Transactions on*, 16(2):113–129, 2005.

[6] William J Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. In *Design Automation Conference, 2001. Proceedings*, pages 684–689. IEEE, 2001.

[7] Kees Goossens and Andreas Hansson. The AEthereal network on chip after ten years: Goals, evolution, lessons, and future. In *Design Automation Conference (DAC)*, pages 306–311. IEEE, 2010.

[8] Yuanfang Hu, Hongyu Chen, Yi Zhu, A Chien, and Chung-Kuan Cheng. Physical synthesis of energy-efficient networks-on-chip through topology exploration and wire style optimization. In *ICCD*, pages 111–118. IEEE, 2005.

[9] Yuanfang Hu, Yi Zhu, Hongyu Chen, Ronald Graham, and Chung-Kuan Cheng. Communication latency aware low power noc synthesis. In *Proceedings of the 43rd annual Design Automation Conference*, pages 574–579. ACM, 2006.

[10] Weichen Liu, Jiang Xu, Xiaowen Wu, Yaoyao Ye, Xuan Wang, Wei Zhang, Mahdi Nikdast, and Zhehui Wang. A NoC traffic suite based on real applications. In *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on*, pages 66–71. IEEE, 2011.

[11] Srinivasan Murali, David Atienza, Luca Benini, and Giovanni De Michel. A multi-path routing strategy with guaranteed in-order packet delivery and fault-tolerance for networks on chip. In *Proceedings of the 43rd annual Design Automation Conference*, pages 845–848. ACM, 2006.

[12] Srinivasan Murali and Giovanni De Micheli. Bandwidth-constrained mapping of cores onto NoC architectures. In *Proceedings of the conference on Design, automation and test in Europe-Volume 2*. IEEE Computer Society, 2004.

[13] Martin Schoeberl, Florian Brandner, Jens Sparsø, and Evangelia Kasapaki. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *Sixth IEEE/ACM International Symposium on Networks on Chip (NoCS)*, pages 152–160. IEEE, 2012.

[14] Charles L Seitz. Let's route packets instead of wires. In *Proceedings of the sixth MIT conference on Advanced research in VLSI*, pages 133–138. MIT Press, 1990.

[15] Rasmus Bo Sørensen, Jens Sparsø, Mark Ruvald Pedersen, and Jaspur Højgaard. A metaheuristic scheduler for time division multiplexed network-on-chip. Technical Report DTU Compute Technical Report-2014-04, Technical University of Denmark, 2014.

[16] Theocharis Theocharides, Gregory M Link, Narayanan Vijaykrishnan, and Mary Jane Irwin. Networks on chip (NoC): Interconnects of next generation systems on chip. *Advances in Computers*, 63:35–89, 2005.